MergeSort is a popular and efficient sorting algorithm that uses the divide-and-conquer strategy. It works as follows:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort each half.
3. **Combine**: Merge the two sorted halves to produce a single sorted array.

Here's a step-by-step breakdown of how MergeSort works:

1. **Base Case**: If the array has one or zero elements, it is already sorted.
2. **Recursive Case**:
   - Divide the array into two roughly equal halves.
   - Recursively apply MergeSort to each half.
   - Merge the two sorted halves to form a single sorted array.

Merging Process

The merging process involves comparing elements from the two halves and arranging them in sorted order. This process is linear in terms of the number of elements being merged.

MergeSort Algorithm

Here's a pseudocode for MergeSort:

```
MergeSort(array):
    if length of array <= 1:
        return array
    mid = length of array // 2
    left_half = MergeSort(array[0:mid])
    right_half = MergeSort(array[mid:])
    return Merge(left_half, right_half)

Merge(left, right):
    result = []
    while left is not empty and right is not empty:
        if left[0] <= right[0]:
            append left[0] to result
            remove left[0] from left
        else:
            append right[0] to result
            remove right[0] from right
    append remaining elements of left (if any) to result
    append remaining elements of right (if any) to result
    return result
```

Mathematical Analysis of Comparisons in MergeSort

To analyze the number of comparisons MergeSort makes, let's break it down:

1. **Splitting the Array**: The array is split $log(n)$ times until we get to subarrays of length 1. This is because each split halves the size of the array.

2. **Merging Subarrays**:

   - Merging two subarrays of length $n/2$ requires at most $n-1$ comparisons.
   - At each level of recursion, we merge subarrays whose total length sums to n.
   - There are $log(n)$ levels in the recursion tree (each split results in two subarrays, which is a binary tree structure).

Total Comparisons

Let $C(n)$ be the total number of comparisons MergeSort makes to sort an array of length n. The recurrence relation for MergeSort comparisons can be expressed as:

$$C(n) = 2C(n/2) + n - 1$$

Where:

- $2C(n/2)$ accounts for the comparisons made in sorting the two halves.
- $n-1$ accounts for the comparisons made during the merge step.

To solve this recurrence relation, we use the method of repeated substitution:

$$C(n) = 2[2C(n/4) + n/2 - 1] + n - 1$$

$$= 4C(n/4) + 2(n/2 - 1) + n - 1$$

$$= 4C(n/4) + n - 2 + n - 1$$

$$= 4C(n/4) + 2n - 3$$

Continuing this expansion:

$$= 8C(n/8) + 3n - 7$$

$$= 16C(n/16) + 4n - 15$$

$$= \cdots$$

$$= 2^k C(n/2^k) + kn - (2^k - 1)$$

Where $k = \log_2 n$, so $n/2^k = 1$:

$$C(n) = 2^{\log_2 n} C(1) + n \log_2 n - (2^{\log_2 n} - 1)$$

$$= nC(1) + n \log_2 n - n + 1$$

$$= n \log_2 n - n + 1$$

Since $C(1) = 0$, we simplify to:

$$C(n) = n \log_2 n - n + 1$$

Thus, the number of comparisons in MergeSort is $O(n \log n)$.

Summary

MergeSort performs at most $O(n \log n)$ comparisons to sort an array of length n. This efficiency comes from the logarithmic depth of the recursion tree and the linear work done at each level of merging.